Ordonnancement dynamique pour un équilibrage de charge quasi-optimal dans les systèmes de traitement de flux

Nicolò Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, Bruno Sericola

Ordonnancement dynamique pour un équilibrage de charge quasi-optimal dans les systèmes de traitement de flux

Nicoló Rivetti[1], Emmanuelle Anceaume[2], Yann Busnel[3,4], Leonardo Querzoni[5] et Bruno Sericola[4]

[1]*Technion - Israel Institute of Technology, Haifa, Israel*
[2]*IRISA / CNRS, Rennes, France*
[3]*IMT Atlantique, Rennes, France*
[4]*Inria Rennes – Bretagne Atlantique, France*
[5]*Sapienza University of Rome, Italy*

La répartition de la charge sur les opérateurs sans état parallélisé dans un système de traitement de flux repose principalement sur le groupement aléatoire des tuples. Chacun de ces derniers peut être assigné à n'importe quelle instance disponible de l'opérateur considéré, indépendamment des assignations précédentes. L'approche classique consiste à transmettre à tour de rôle les tuples aux différentes instances parallèles existantes. Cette politique convient bien tant que le temps d'exécution de tous les tuples est plus ou moins le même. Cette hypothèse est cependant rarement vérifiée en pratique, où les temps d'exécution reposent principalement sur le contenu des tuples, et peut causer un déséquilibre imprévisible menant *in fine* à un accroissement indésirable des temps d'exécution et potentiellement à la défaillance du système. Dans cet article, nous proposons Online Shuffle Grouping (OSG), une solution de groupement permettant de réduire le temps d'exécution global des tuples. OSG commence par estimer, par l'utilisation d'agrégats, la durée d'exécution de chaque tuple, avec des taux d'erreur d'approximation faibles et bornés, lui permettant d'effectuer un ordonnancement pro-actif en temps-réel. Nous proposons une analyse probabiliste de OSG et évaluons son impact sur des applications de traitement de flux, en terme de robustesse et de fiabilité, par une large expérimentation sur la plateforme Microsoft Azure.

Mots-clefs : Traitement de flux; Groupement de clé; Equilibrage de charge; Algorithme d'approximation probabiliste; Evaluation de performance

Travaux co-financés par le projet ANR SocioPlug (ANR-13-INFR-0003) et le projet DeSceNt du Labex CominLabs (ANR-10-LABX-07-01).

1 Introduction

Stream processing systems are today gaining momentum to perform analytics on continuous data streams. Their ability to achieve sub-second latencies, coupled with their scalability, makes them the preferred choice for many big data companies. A stream processing application is commonly modeled as a direct acyclic graph where data operators, represented by vertices, are interconnected by streams of tuples containing data, the directed edges. Scalability is usually attained at the deployment phase parallelizing data operator using multiple instances, each of which will handle a subset of the tuples conveyed by the operator's ingoing stream. The strategy used to route tuples in a stream toward the instances of the receiving operator is called *grouping* function. Operator parallelization is straightforward for *stateless* operators, *i.e.*, data operators whose output is only a function of the current tuple in input. In this case, the grouping function is free to assign the next tuple in the input stream, to any instance of the operator (contrarily to stateful operators, where tuple assignment is constrained). Such grouping functions are often called *shuffle grouping* and are a fundamental element of a number of stream processing applications. Shuffle grouping implementations are

designed to balance the load on the receiving operator instances, increasing the system efficiency.Notable implementations, such as Apache Storm, leverage a simple Round-Robin scheduling strategy routing the same number of tuples to each instance. Amini *et al.* [1], look into non-uniform operators and/or tasks. These approaches are effective as long as the time taken by the operator instance to process a tuple (tuple *execution time*) is the same for any tuple. In this case, all parallel instances of the operator experience over time, on average, the same load. However,for many practical use cases we cannot assume that all tuples of a stream have the same execution time. The tuple execution time, in fact, may depend on the tuple content itself. This is often the case whenever the receiving operator implements a logic with branches where only a subset of the incoming tuples travels through each single branch, each generating different loads. Then the execution time will change from tuple to tuple. In this case shuffle grouping implemented with Round-Robin may produce imbalance (an example is given in [4]), as some tuple may end-up being queued on some overloaded operator instances, increasing the time needed for a tuple to be completely processed by the application (tuple *completion time*).

We introduce *Online Shuffle Grouping* (OSG) a novel approach to shuffle grouping that aims at reducing tuple completion times by carefully scheduling each incoming tuple. In particular, OSG makes use of sketches to efficiently keep track of tuple execution times at the available operator instances and then applies a greedy online multiprocessor scheduling algorithm to assign tuples to operator instances at runtime. The status of each instance is monitored in a smart way in order to detect possible changes in the input load distribution and coherently adapt the scheduling. As a result, OSG provides an important performance gain in terms of tuple completion times with respect to Round-Robin for all those settings where tuple processing times are not similar, but rather depend on the tuple content. In summary, we provide the following contributions: (**i**) We present OSG, the first solution (to the best of our knowledge) to load balance shuffle grouped parallel operator (non-uniform) with non-uniform tuple execution times; (**ii**) We analyze study the two components of our solution; (**iii**) We evaluate OSG with an extensive evaluation.

2 System Model

We consider a stream processing system (SPS) deployed on a cluster where several computing nodes exchange data through messages over a network. The SPS executes a stream processing application represented by a *topology*: a directed acyclic graph interconnecting operators, represented by nodes, with data streams (DS), represented by edges. Each topology contains at least a *source*, *i.e.,* an operator connected only through outbound DSs, and a *sink*, *i.e.,* an operator connected only through inbound DSs. For the sake of clarity, and without loss of generality[†], we consider a topology with an operator S (*scheduler*) which schedules the tuples that are consumed by the k instances $O_1, \cdots, O_k$ of operator O. Each operator instance has a FIFO input queue where tuples are buffered while the instance is busy processing previous tuples. Tuples are assigned to sub-streams through *shuffle grouping*, where each incoming tuple can be assigned to any sub-stream. Data injected by the source is encapsulated in units called tuples and each data stream is a sequence of tuples whose size (that is the number of tuples) m is unknown. To simplify the discussion, in the rest of this work we deal with streams of unary tuples with a single non negative integer value. We denote by $w_{t,op}$ the execution time of tuple t on operator instance O_{op} (O_{op} denotes a generic operator instance) The execution time $w_{t,op}$ is modeled as an unknown function[‡] of the content of tuple t and that may be different for each operator instance (*i.e.,* operator instances may be non-uniform). We assume that $w_{t,op}$ depends on a single fixed and known attribute value of t. The probability distribution of such attribute values, as well as $w_{t,op}$, are unknown and may change over time. However, The time frame between two subsequent changes allows the algorithm to adapt. Abusing the notation, we may omit in $w_{t,op}$ the operator instance identifier subscript. The goal we target is to minimize the average tuple completion time $\overline{L}$.

[†] The case where operator S is parallelized is discussed in [4].

[‡] In the experimental evaluation we relax the model by taking into account the execution time variance

3 Online Shuffle Grouping

Online Shuffle Grouping is a shuffle grouping implementation based on a simple, yet effective idea: if we know the execution time $w_{t,op}$ of each tuple t on any operator instances, we can schedule the execution of incoming tuples minimizing the average per tuple completion time at the operator instances. However, the value of $w_{t,op}$ is generally unknown. A common solution is to build a cost model for the tuple execution time and then use it to proactively schedule incoming load. However building an accurate cost model usually requires a large amount of *a priori* knowledge on the system. Furthermore, once a model has been built, it can be hard to handle changes in the system or input stream characteristics at runtime. To overcome all these issues, OSG takes decisions based on the estimation $\widehat{\mathcal{C}}_{op}$ of the execution time assigned to instance O_{op}, that is $\mathcal{C}_{op} = \sum_{t \in O_{op}^{in}} w_{t,op}$. In order to do so, OSG computes an estimation $\widehat{w}_{t,op}$ of the execution time $w_{t,op}$ of each tuple t on each operator instance O_{op}. Then, OSG can also compute the sum of the estimated execution times of the tuples assigned to an instance O_{op}, *i.e.,* $\widehat{\mathcal{C}}_{op} = \sum_{t \in O_{op}^{in}} \widehat{w}_{t,op}$, which in turn is the estimation of $\mathcal{C}_{op}$. A Greedy Online Scheduler algorithm for multiprocessor scheduling is then fed with estimations for all the available operator instances. Online scheduling means that the scheduler does not know in advance the sequence of tasks it has to schedule. In [4] prove that Greedy Online Scheduler approximates an optimal omniscient scheduling algorithm, that is an algorithm that knows in advance all the tuples it will received.[§]

To implement this approach, each operator instance builds a sketch (*i.e.,* a memory efficient data structure) that will track the execution time of the tuples it processes. When a change in the stream or instance(s) characteristics affects the tuples execution times on some instances, the concerned instance(s) will forward an updated sketch to the scheduler which will then be able to (again) correctly estimate the tuples execution times. This solution does not require any *a priori* knowledge on the stream composition or the system, and is designed to continuously adapt to changes in the input distribution or on the instances load characteristics. In addition, this solution is *proactive*, namely its goal is to avoid unbalance through scheduling, rather than detecting the unbalance and then attempting to correct it. A *reactive* solution can hardly be applied to this problem, in fact it would schedule input tuples on the basis of a previous, possibly stale, load state of the operator instances. In addition, reactive scheduling typically imposes a periodic overhead even if the load distribution imposed by input tuples does not change over time. For the sake of clarity, we consider a topology with a single operator S (*i.e.,* a *scheduler*) which schedules the tuples of a DS O^{in} consumed by the k instances of operator O (*cf.,* Figure 1). To encompass topologies where the operator generating DS O^{in} is itself parallelized, we can easily extend the model by taking into account parallel instances of the scheduler S. We show [4] that also in this setting OSG performances are better than Round-Robin scheduling policy. Our approach is hop-by-hop, *i.e.,* we consider a single shuffle grouped edge in the topology at a time. However, OSG can be applied to any shuffle grouped stage of the topology.

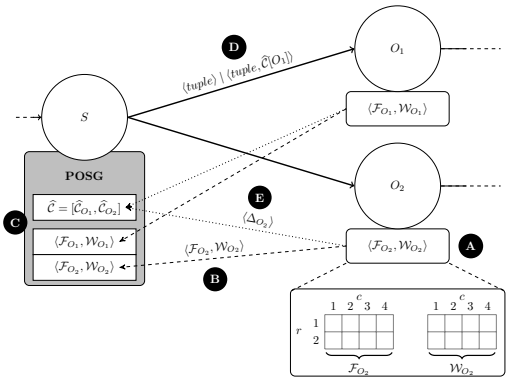

Fig. 1: Online Shuffle Grouping design with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$) and $k = 2$.

3.1 OSG design

Each operator instance *op* maintains two `Count Min` [2] sketch matrices (Figure 1.A): the first one, denoted by $\mathcal{F}_{op}$, tracks the tuple frequencies $f_{t,op}$; the second, denoted by $\mathcal{W}_{op}$, tracks the tuples cumulated execution times $W_{t,op} = w_{t,op} \times f_{t,op}$. Both `Count Min` matrices have the same sizes and hash functions. The latter is the generalized version of the `Count Min` presented in [2] where the update value is the tuple execution time when processed by the instance (*i.e.,* $w_{t,op}$). The operator instance will update both matrices after each tuple execution. If the operator instance detects that there has been a change in the tuples execution time, then it forwards (Figure 1.B) the updated matrices to the scheduler. The scheduler (Figure 1.C)

[§] Notice that this is a variant of the join-shortest-queue (JSQ) policy, where we measure the queue length as the time needed to execute all the buffered tuples, instead of the number of buffered tuples.

maintains the estimated cumulated execution time for each instance, in a vector $\widehat{\mathcal{C}}$ of size k, and the set of pairs of matrices: $\{\langle \mathcal{F}_{op}, \mathcal{W}_{op} \rangle\}$, initially empty. For each tuple t, the scheduler assigns it to the operator instance applying the Greedy Online Scheduler algorithm, *i.e.,* assigns the tuple to the operator instance with the least estimated cumulated execution time, *i.e.,* routes the current tuples towards operator op^*, where $op^* = \min \arg_{op \in [k]} \widehat{\mathcal{C}}[op]$. Then it increments the target instance estimated cumulated execution time with the estimated tuple execution time $\widehat{w}_{t,op^*}$, *i.e.,* $\widehat{\mathcal{C}}[op^*] \leftarrow \widehat{\mathcal{C}}[op^*] + \widehat{w}_{t,op^*}$. There is a delay between any change in the stream or operator instances characteristics and when the scheduler receives the updated $\mathcal{F}_{op}$ and $\mathcal{W}_{op}$ matrices from the affected operator instance(s). This introduces a skew in the cumulated execution times estimated by the scheduler. In order to compensate for this skew, we introduce a synchronization mechanism (Figure 1.D and 1.E) that springs whenever the scheduler receives a new pair of matrices from any operator instance. Notice also that there is an initial transient phase in which the scheduler has not yet received any information from operator instances. This means that in this first phase it has no information on the tuples execution times and is forced to use the Round-Robin policy. The synchronization mechanism is thus also needed to initialize the estimated cumulated execution times in this initial transient phase. The complete theoretical analysis of OSG is available in [4].

4 Experimental Evaluation

We extensively tested OSG both in a simulated environment with synthetic datasets and on a prototype implementation running with real data. Due to space constraints, the complete results are available in [4]. In this test we want to compute the *reach* of twitted terms in a dataset generated through LDBC Social Network Benchmark [3]. Using the default parameters of this benchmark, we obtained a followers graph of $9,960$ nodes and $183,005$ edges (the maximum out degree was 734) as well as a stream of $2,114,269$ tweets where the most frequent author has an empirical probability of occurrence equal to 0.0038. The reach of a term is the total number of estimated unique Twitter users to which were delivered tweets about the search term. Usually, this metric is calculated through a periodic batch process using the followers graph, where edges are enriched with re-tweet probabilities. We propose instead to compute this value in a streaming fashion, for each tweet, restricting the computation to a depth of 3 in the followers graph of $9,960$ nodes. Globally, the tuple execution times belong to the interval $[0.01, 70]$ ms, the most frequent tuple execution time is in average 65 ms, while the average per tuple execution time is 20 ms. Figure 2 shows the mean, maximum and minimum average completion time $\overline{L}^{alg}$ for both OSG and ASSG (Apache Storm standard Shuffle Grouping implementation) as a function of the number of instances k over 10 executions. Except for the unanticipated spike of ASSG for $k = 5$, the completion latency decreases as k increases. For all k OSG has a smaller mean average completion latency than ASSG. In addition, for most values of k, the maximum average completion latency of OSG is smaller or equal to the minimum average completion latency of ASSG. Finally, the average speed up of OSG with respect to ASSG is at least 1.05, at most 3.4 and in average 1.5. To achieve these results, OSG exchanges only a few thousand additional messages, against a stream size of $m = 2,114,269$.

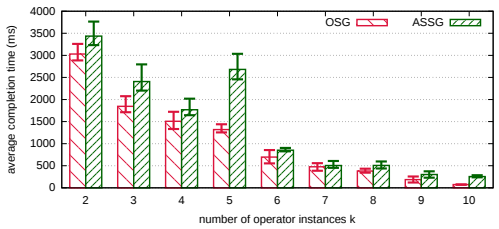

Fig. 2: Prototype average per tuple completion time $\overline{L}^{alg}$ as a function of the number of operators k.

References

[1] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *Proc. of the 26th IEEE Int. Conf. on Dist. Comp. Syst.*, 2006.

[2] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. of Algorithms*, 2005.

[3] Linked Data Benchmark Council. Social Network Benchmark. `http://ldbcouncil.org/benchmarks/snb`.

[4] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola. Online scheduling for shuffle grouping in distributed stream processing systems. In *Proc. of the 17th Int. Middleware Conf.*, 2016.